ON THE COST OF TYPE-TAG SOUNDNESS

Ben Greenman    Zeina Migeed
ON THE COST OF TYPE-TAG SOUNDNESS

1. Tag soundness
2. Performance cost of soundness
3. Evaluation method
4. Conclusions
TYPE-TAG SOUNDNESS
Type Soundness

If $\vdash e : \tau$ then either:

- $e \rightarrow^* v$ and $\vdash v : \tau$
- $e$ diverges
- $e \rightarrow^* \text{Error}$ (division by zero, etc.)

No undefined behavior
Type-based reasoning
Type Soundness

If $ \vdash e : \tau$ then either:

- $e \rightarrow^{*} v$ and $\vdash v : \tau$
- $e$ diverges
- $e \rightarrow^{*} \text{Error}$ (division by zero, etc.)
Tag Soundness

If \( \vdash e : \tau \) then either:

- \( e \rightarrow^* v \) and \( \vdash v : \lfloor \tau \rfloor \)
- \( e \) diverges
- \( e \rightarrow^* \text{Error} \) (division by zero, etc.)
Tag Soundness

If $\vdash e : \tau$ then either:

- $e \rightarrow^* v$ and $\vdash v : \llbracket \tau \rrbracket$
- $e$ diverges
- $e \rightarrow^* \text{Error}$ (division by zero, etc.)

$\llbracket \tau \rrbracket = K$

$\llbracket \text{Int} \rrbracket = \text{Int}$

$\llbracket \tau \times \tau' \rrbracket = \text{Pair}$

$\llbracket \tau \rightarrow \tau' \rrbracket = \text{Fun}$

...
Tag Soundness

If \( \vdash e : \tau \) then either:

1. \( e \xrightarrow{*} v \) and \( \vdash v : \llbracket \tau \rrbracket \)
2. \( e \) diverges
3. \( e \xrightarrow{*} \text{Error} \)

\[ \llbracket \tau \rrbracket = K \]
\[ \llbracket \text{Int} \rrbracket = \text{Int} \]
\[ \llbracket \tau \times \tau' \rrbracket = \text{Pair} \]
\[ \llbracket \tau \to \tau' \rrbracket = \text{Fun} \]

No undefined behavior
Tag-based reasoning
Types vs. Tags

If \( \vdash e : \text{Int} \times \text{Int} \) and \( e \rightarrow^* v \) then \( v \) might be:

<table>
<thead>
<tr>
<th>Type Soundness</th>
<th>Tag Soundness</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>(&quot;A&quot;, 0)</td>
</tr>
<tr>
<td>(3, 2)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(-7, 9)</td>
<td>(3, 2)</td>
</tr>
<tr>
<td></td>
<td>(-7, 9)</td>
</tr>
<tr>
<td></td>
<td>(0, (1, 2))</td>
</tr>
</tbody>
</table>
If $\vdash e : \text{Int} \times \text{Int}$ and $e \xrightarrow{*} v$ then $v$ might be:

<table>
<thead>
<tr>
<th>Type Soundness</th>
<th>Tag Soundness</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(0, 0)$</td>
<td>$(&quot;A&quot;, 0)$</td>
</tr>
<tr>
<td>$(3, 2)$</td>
<td>$(0, 0)$</td>
</tr>
<tr>
<td>$(-7, 9)$</td>
<td>$(3, 2)$</td>
</tr>
<tr>
<td></td>
<td>$(-7, 9)$</td>
</tr>
<tr>
<td></td>
<td>$(0, (1, 2))$</td>
</tr>
<tr>
<td></td>
<td>fast</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>Type Sound?</strong></td>
<td>✗</td>
</tr>
<tr>
<td><strong>Tag Sound?</strong></td>
<td>✓</td>
</tr>
</tbody>
</table>
PERFORMANCE COST OF SOUNDNESS
Problem: Safe Interaction

e^{\tau} \rightarrow e^{\tau} \rightarrow e^{\tau} \rightarrow \tau

\text{?}
Gradual Typing

e^\tau \rightarrow e^\tau \rightarrow e^\tau \rightarrow \tau

\lambda
User Input

$e^\tau \xrightarrow{*} \text{read()} \xrightarrow{\tau} \tau$

Enter a value:

>
Deserialization

e^{\tau} \xrightarrow[*] \text{unzip}()^{\tau} \xrightarrow[] \tau

\begin{array}{c}
0110 \\
1110 \\
1011
\end{array}
Primitive Operations (δ)

\[ e^{\text{Int}} \xrightarrow{*} v + v^{\text{Int}} \xrightarrow{} \text{Int} \]

\[ E[v,v] \xrightarrow{} \ldots \xrightarrow{} v' \]
Unreliable Source

Enter a value:

Enter a value:

\[ \tau \]

\[ e^{\tau} \]

E[\cdot]

0110
1110
1011
Option 1: Trust
Option 2: Check

\[ \begin{align*}
\text{e}^\tau & \rightarrow \text{v}^\tau \\
\text{v}^\tau & \rightarrow \ldots \rightarrow \text{v}^\tau \\
\end{align*} \]
Option 2: Check

\[ e^\tau \rightarrow v^\tau \rightarrow \ldots \rightarrow v^\tau \]

\text{COST OF SOUNDNESS}
Cost of Types \((\longrightarrow^* \text{ slow})\)
Cost of Tags \( \leftrightarrow_{\text{fast}} \)

\[(6,1) \quad \text{Int} \times \text{Int} \quad (6,1) \quad \text{Int} \times \text{Int} \]
COST OF SOUNDNESS IN RETICULATED
Retic vs. Python
def dist(pt : Tuple(Int,Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```python
def dist(pt : Tuple(Int,Int)) -> Int:
x = pt[0]
y = pt[1]
return abs(x + y)
```

```
dist((0, 0))  →  0
```
Reticulated

```python
def dist(pt : Tuple(Int, Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```

dist("NaN")  ➞  Expected Tuple
Reticulated

def dist(pt : Tuple(Int,Int)) -> Int:
x = pt[0]
y = pt[1]
return abs(x + y)

dist((0, "NaN")) → Expected Int
Evaluation Method
1. Fully-Typed
2. Configurations
3. Measure

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7s</td>
<td>9s</td>
<td>2s</td>
</tr>
<tr>
<td>5s</td>
<td>24s</td>
<td>9s</td>
</tr>
<tr>
<td>14s</td>
<td>5s</td>
<td>21s</td>
</tr>
<tr>
<td>9s</td>
<td>6s</td>
<td>9s</td>
</tr>
<tr>
<td>8s</td>
<td>4s</td>
<td>5s</td>
</tr>
</tbody>
</table>
What % have at most Dx overhead?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7s</td>
<td>9s</td>
<td>2s</td>
</tr>
<tr>
<td>5s</td>
<td>24s</td>
<td>9s</td>
</tr>
<tr>
<td>14s</td>
<td>5s</td>
<td>21s</td>
</tr>
<tr>
<td>9s</td>
<td>6s</td>
<td>9s</td>
</tr>
<tr>
<td>8s</td>
<td>4s</td>
<td>5s</td>
</tr>
</tbody>
</table>
What % have at most $D_x$ overhead?

$D = 4$, vs.
Evaluation Method

$2^2$ or $O(2)$
EXPERIMENT & RESULTS
## Benchmarks

<table>
<thead>
<tr>
<th>DLS 2014</th>
<th>POPL 2017</th>
<th>PEPM 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>futen</td>
<td>call_method</td>
<td>espionage</td>
</tr>
<tr>
<td>http2</td>
<td>call_simple</td>
<td>pythonflow</td>
</tr>
<tr>
<td>slowSHA</td>
<td>chaos</td>
<td>take5</td>
</tr>
<tr>
<td>aespython</td>
<td>fannkuch</td>
<td>sample_fsm</td>
</tr>
<tr>
<td>stats</td>
<td>go</td>
<td></td>
</tr>
<tr>
<td></td>
<td>meteor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nbody</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nqueens</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pidigits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pystone</td>
<td></td>
</tr>
<tr>
<td></td>
<td>spectralnorm</td>
<td></td>
</tr>
</tbody>
</table>
# Typed Components

<table>
<thead>
<tr>
<th>DLS 2014</th>
<th>POPL 2017</th>
<th>PEPM 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>34 *</td>
<td>1</td>
<td>19 *</td>
</tr>
<tr>
<td>79 *</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Exhaustive Results

What % of configurations have at most 4x overhead?

![Bar chart showing 50% of configurations have 4x overhead.](chart.png)
Exhaustive Results

What % of configurations have at most $D_x$ overhead?

- 100%
- 50%
- 4x
- 2x
- 6x
- 8x
Exhaustive Results

What % of configurations have at most $Dx$ overhead?

![Graph showing the percentage of configurations with at most $Dx$ overhead. The x-axis represents different multiples of $x$ (1x, 2x, 4x, 6x, 8x), and the y-axis shows the percentage (0%, 50%, 100%). The graph indicates that over 50% of configurations have at most 4x overhead.]
Approximate Results

What % of configurations have at most $4x$ overhead, based on $R$ samples of $S$ configurations each?

100%

50%

4x
Approximate Results

What % of configurations have at most $Dx$ overhead, based on $R$ samples of $S$ configurations each?
Approximate Results

What % of configurations have at most $Dx$ overhead, based on $R$ samples of $S$ configurations each?
Espionage: 4,096 configurations

Aespython: 10 samples of 340 configurations
Cost of Tag Soundness

• Worst-case overhead: under 10x
This is an APPLES to ORANGES comparison!
Cost of Tag Soundness

- Worst-case overhead: under 10x
- Best-case overhead: 1x -- 4x
  - adding types never* improves performance
- Slowest configuration: fully-typed
  - Overhead $\propto$ number of type annotations
Runtime vs. # Types

![Graph showing spectral norm vs. runtime](image)
Runtime vs. # Types
Speedup?

- Unsound optimization for read-only values (tuples)
- Double-checks method calls
Runtime vs. # Types

- futen: 1,310,720 points
- http2: 640 points
- slowSHA: 5,242,880 points
- call_method: 5,120 points
- call_simple: 2,560 points
- chaos: 1,310,720 points
- lannkuch: 80 points
- float: 2,560 points
- go: 5,120 points
- meteor: 10,240 points
- nbody: 1,280 points
- nqueens: 160 points
- pidigits: 1,280 points
- pystone: 655,360 points
- spectralnorm: 1,280 points
- Espionage: 163,840 points
- PythonFlow: 163,840 points
- take5: 2,621,440 points
- sample_fsm: 76,000 points
- aespython: 136,000 points
- stats: 316,000 points
Experiment

- granularity: functions & class-fields
- 10 samples of [10 * (F + C)] configurations
- Karst at Indiana University cluster (32GB RAM, 250GB other)
- Reticulated, master branch, commit e478343
- Python 3.4.3
- 40 iterations per configuration, report average
- 200 values of D on x-axis
Figure 11. Runtime comparison of Reticulated Python to standard Python 3.4. Experiments were performed on an Ubuntu 14.04 laptop with a 2.8GHz Intel i7-3840QM CPU and 16GB memory.
Module **Marshal**

```ocaml
module Marshal: sig .. end
```

Marshaling of data structures.

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of OCaml.

**Warning:** *marshaling is currently not type-safe.* The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the `Marshal.from_` functions is given as `'a`, but this is misleading: the returned OCaml value does not possess type `'a` for all `'a`; it has one, unique type which cannot be determined at compile-time. The programmer should explicitly give the expected type of the returned value, using the following syntax:

- `(Marshal.from_channel chan : type)`. Anything can happen at run-time if the object in the file does not belong to the given type.
References

• Vitousek, Swords, Siek. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. POPL 2017

UNUSED SLIDES
Takikawa Method

- granularity
- experimental modules, fixed modules
- configurations
- baseline
- performance ratio